# Dynamic slicing for Concurrent Constraint Languages

**Moreno Falaschi**
*Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche*
*Università di Siena, Italy.*
*moreno.falaschi@unisi.it*

**Maurizio Gabbrielli**
*Dipartimento di Informatica - Scienza e Ingegneria*
*Università di Bologna, Italy.*
*maurizio.gabbrielli@unibo.it*

**Carlos Olarte**
*ECT, Universidade Federal do Rio Grande do Norte, Brazil*
*carlos.olarte@gmail.com*

**Catuscia Palamidessi**
*INRIA and LIX, École Polytechnique, France*
*catuscia@lix.polytechnique.fr*

**Abstract.** Concurrent Constraint Programming (CCP) is a declarative model for concurrency where agents interact by telling and asking constraints (pieces of information) in a shared store. Some previous works have developed (approximated) declarative debuggers for CCP languages. However, the task of debugging concurrent programs remains difficult. In this paper we define a dynamic slicer for CCP (and other language variants) and we show it to be a useful companion tool for the existing debugging techniques. We start with a partial computation (a trace) that shows the presence of bugs. Often, the quantity of information in such a trace is overwhelming, and the user gets easily lost, since she cannot focus on the sources of the bugs. Our slicer allows for marking part of the state of the computation and assists the user to eliminate most of the redundant information in order to highlight the errors. We show that this technique can be tailored

Address for correspondence: Moreno Falaschi, Università di Siena, Italy. e-mail: moreno.falaschi@unisi.it.

to several variants of CCP, such as the timed language ntcc, linear CCP (an extension of CCP-based on linear logic where constraints can be consumed) and some extensions of CCP dealing with epistemic and spatial information. We also develop a prototypical implementation freely available for making experiments.

**Keywords:** Concurrent Constraint Programming, Program slicing, Debugging

# 1. Introduction

Concurrent constraint programming (CCP) [1, 2] (see a survey in [3]) combines concurrency primitives with the ability to deal with constraints, and hence, with partial information. The notion of concurrency is based upon the shared-variables communication model. CCP is intended for reasoning, modeling and programming concurrent agents (or processes) that interact with each other and their environment by posting and asking information in a medium, a so-called store. Agents in CCP can be seen as both computing processes (behavioral style) and as logic formulas (declarative style). Hence CCP can exploit reasoning techniques from both process calculi and logic.

CCP is a very flexible model and then, it has been applied to an increasing number of different fields such as probabilistic and stochastic [4], timed [5, 6, 7] and mobile [8] systems. More recently, CCP languages have been used for the specification of spatial and epistemic behaviors as in, e.g., social networks [9, 10].

One crucial problem when working with a concurrent language is being able to provide tools to debug programs. This is particularly useful for a language in which a program can generate a large number of parallel running agents. In order to tame this complexity, abstract interpretation techniques have been considered (e.g. in [11, 12, 13]) as well as (abstract) declarative debuggers following the seminal work of Shapiro [14]. However, these techniques are approximated (case of abstract interpretation) or it can be difficult to apply them when dealing with complex programs (case of declarative debugging). It would be useful to have a semi automatic tool able to interact with the user and filter, in a given computation, the information which is relevant to a particular observation or result. In other words, the programmer could mark the outcome that she is interested to check in a given computation that she suspects to be wrong. Then, a corresponding depurated partial computation is obtained automatically, where only the information relevant to the marked parts is present.

Slicing was introduced in some pioneer works by Mark Weiser [15]. It was originally defined as a static technique, independent of any particular input of the program. Then, the technique was extended by introducing the so called dynamic program slicing [16]. This technique is useful for simplifying the debugging process, by selecting a portion of the program containing the faulty code. Dynamic program slicing has been applied to several programming paradigms, for instance to imperative programming [16], functional programming [17], Term Rewriting [18], and functional logic programming [19]. The reader may refer to [20] for a survey.

In this paper we present the first formal framework for CCP dynamic slicing and show, by some working examples and a prototypical tool, the main features of this approach. Our aim is to help the programmer to debug her program, in cases where she could not find the bugs by using other debuggers. We proceed with three main steps. First we extend the standard operational semantics of CCP

to a "enriched semantics" that adds the needed information for the slicer. Second, we propose several analyses of the faulty situation based on error symptoms, including causality, variable dependencies, unexpected behaviors and store inconsistencies. Thirdly, we define a marking algorithm of the redundant items and define a trace slice. Our algorithm is flexible and we show that it can deal with different extensions of CCP.

This paper is an extended version of [21]. We refine several technical details, we add many more explanations, and present full proofs. From the theoretical point of view, we extend the results in [21] as follows:

- we consider in a general and uniform way different variants and extensions of CCP not considered previously in [21]. In particular, we extend our framework in order to deal with epistemic and spatial extensions of CCP (`eccp` and `sccp`) and the resource conscious version of CCP (linear CCP –`lcc`–);
- we state formally the requirements needed on the constraint system for the definition of the slicer;
- we allow also to mark processes (and not only constraints) and propose two additional marking techniques in Section 3.2 that turn out to be interesting for analyzing `lcc` programs;
- causality relation among processes is better captured now as shown in Example 3.9.

*Organization.* Section 2 describes CCP and its operational semantics. We also briefly describe the other concurrent constraint languages considered here: `lcc`, `eccp`, `sccp`, and `ntcc`. The operational semantics of these languages, which are extensions of CCP, is shown by adding appropriate rules to the semantics for CCP. We present a table of rules which allows to obtain in a simple and direct way the semantics of each of the considered languages. In Section 3 we introduce a slicing technique for CCP and its extensions. We also present a prototypical implementation of the slicer available at `http://subsell.logic.at/slicer/` and some illustrative programs and examples. Other detailed examples can be found in the web page of the tool as, for instance, a biochemical system specified in timed CCP. Finally, Section 4 concludes.

## 2. Concurrent Constraint Programming

Processes in CCP *interact* with each other by *telling* and *asking* constraints (pieces of information) in a common store of partial information. The type of constraints is not fixed but parametric in a constraint system (CS). Intuitively, a CS provides a signature from which constraints can be built from basic tokens (e.g., predicate symbols), and two basic operations: conjunction ($\sqcup$) and variable hiding ($\exists$). As usual, the variable $x$ is bound in $\exists x.c$. Given a set of variables $X = \{x_1, ..., x_n\}$, we use $\exists X.c$ to denote $\exists x_1. \cdots .\exists x_n.c$. The CS defines also an *entailment* relation ($\models$) specifying inter-dependencies between constraints: $c \models d$ means that the information $d$ can be deduced from the information $c$ (for instance, $x > 42 \models x > 0$). It is also customary to consider two distinguished constraints: `t` (denoting true) representing the least token of information and `f` (denoting false) representing inconsistency. Hence, $\texttt{f} \models c$ for any $c$. We shall write $c \equiv d$ if $c \models d$ and $d \models c$.

Constraint systems can be formalized in different ways and the structure of the CS determines the behavior of the resulting CCP language. For instance, the general construction of CS as a Scott information system as in [2] or as cylindric algebras [22] (see also [13]) allows for defining typical CSs as the Herbrand constraint system underlying logic programming, the Kahn Constraint System underlying data-flow languages, the Rational Interval Constraint System [2] and the finite domain constraint system (FD) [23]. By adding *space functions* (topological and closure operators), it is possible to specify epistemic and spatial information leading to languages such as `eccp` and `sccp` [9]. Finally, the notion of CS can be also set up in a suitable fragment of logic [24, 25]. If a substructural logic as linear logic [26] is considered, then the resulting CS allows agents to produce and also consume tokens of information as in linear CCP (`lcc`) [27].

Our framework is general enough to deal with any concrete instance of a CS that provides a suitable meaning to the basic tokens from which constraints are built and to the symbols $\mathtt{t}, \mathtt{f}, \models, \exists, \sqcup$. We only need to assume that the CS offers facilities to define the following functions.

**Remark 2.1. (Helper functions)**
Let $S$ be a (multi)set of constraints and $c$ be a constraint and assume that $\bigsqcup S \models c$. For our analyses, we assume that the (implementation of the) CS offers the following helper functions:

- $bv(c)$, returning the bound variables of $c$;
- $S_{min}(S, c) = \bigcup\{S' \subseteq S \mid \bigsqcup S' \models c \text{ and } S' \text{ is set minimal}\}$ where "$S'$ is set minimal" means that for any $S'' \subset S'$, $S'' \not\models c$; and
- $basic(c)$, returning the (multi)set of basic atoms/tokens of $c$, e.g.,

$$basic(c) = \begin{cases} c \text{ if } c \text{ is an atomic proposition, } \mathtt{t}, \mathtt{f} \\ basic(c') \text{ if } c = \exists x.c' \\ basic(c_1) \cup basic(c_2) \text{ if } c = c_1 \sqcup c_2 \end{cases}$$

The function $basic(c)$ decomposes $c$ in its atomic components. This will allow us to highlight, in a more precise way, the exact contributions of each process to produce a final store of interest for the user. For instance, if a process adds $c \sqcup d$ to the store but only $c$ is important for the user, our technique will be able to produce "$c \sqcup \bullet$" where $\bullet$ means "irrelevant" (this will be later formalized in Notation 1). Since $basic(c)$ is a simple inductive definition on the structure of constraints, we think that this is a mild requirement for any CS.

The function $S_{min}(S, c)$ will allow us to identify the set of relevant constraints in $S$ that entail $c$. We note that "set minimality", in general, can be expensive to compute. However, we believe that in some practical cases, as shown in the examples in Section 3, this is not so heavy. In any case, we can always use supersets of the minimal ones which are easier to compute but less precise for eliminating useless information. For instance, in the FD and the Herbrand constraint system, a good approximation is to start with $S_c \subseteq S$ containing the constraints that share variables with the free variables in $c$. Then, systematically add to $S_c$ constraints from $S$ that share variables with the constraints already in $S_c$. As another example, some CSs require only synchronization constraints that are set of atomic propositions without non logical axioms [28]. Then, it is easy to compute $S_{min}(S, c)$

that will contain exactly the same atoms occurring in $c$. Finally, we note that checking entailments is the main task of a CS. Then, particular implementations may generate some traces when testing $\bigsqcup S \models c$ that can be useful to compute $S_{min}(S, c)$.

**Remark 2.2.** Let $c$ be a constraint and assume by $\alpha$-conversion (i.e., replacing bound names with others not in use) that the bound variables in $c$ are all different and disjoint from the set of free variables in $c$. Then, it is easy to see that the following equivalence holds: $c \equiv \exists bv(c).\ \bigsqcup_{d \in basic(c)} d$.

**Remark 2.3. (Linear and non-linear constraint systems)**
In all the constraint systems discussed in this paper, except for the linear constraint system [27] of `lcc` (Section 2.2.1), $c = c \sqcup c$. Then, we treat $basic(c)$ as a set of atomic constraints. In the case of `lcc`, $basic(c)$ must be understood as a multiset of atomic constraints (where cardinality matters).

## 2.1. The language of CCP processes

We shall start with the basic set of process constructs that constitutes the core of any CCP-based language. Then, we shall introduce new constructs leading to languages as epistemic (`eccp`), spatial (`sccp`), linear (`lcc`) and timed (`ntcc`) CCP.

A typical CCP process language is equipped with: a *tell* operator adding new information (constraints) to a common store (i.e., a set or conjunction of constraints); an *ask* operator querying if a constraint can be deduced from the store; *parallel composition* combining processes concurrently; and a *hiding* operator (also called *restriction* or *locality*) introducing local variables. Additionally, infinite computations can be described by means of *recursion*.

**Definition 2.4. (Syntax of indeterminate CCP agents [2])**
Agents in CCP are built from constraints in the underlying constraint system and the syntax:

$$P, Q ::= \mathbf{skip} \mid \mathbf{tell}(c) \mid \sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i \mid P \parallel Q \mid (\mathbf{local}\, x)\, P \mid p(\overline{x})$$

The process $\mathbf{skip}$ represents inaction. The process $\mathbf{tell}(c)$ adds $c$ to the current store $d$ producing the new store $c \sqcup d$. Given a non-empty finite set of indexes $I$, the process $\sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i$ non-deterministically chooses $P_k$ for execution if the store entails $c_k$. The chosen alternative, if any, precludes the others. This provides a powerful synchronization mechanism based on constraint entailment. When $I$ is a singleton, we shall omit the "$\sum$" and we simply write $\mathbf{ask}\ (c)\ \mathbf{then}\ P$.

The process $P \parallel Q$ represents the parallel (interleaved) execution of $P$ and $Q$. The process $(\mathbf{local}\, x)\, P$ behaves as $P$ and binds the variable $x$ to be local to it. We use $fv(P)$ (resp. $bv(P)$) to denote the set of free (resp. bound) variables in $P$.

The process $p(\overline{x})$ represents a process (or procedure) call. We assume that there is a unique definition for $p$ of the form $p(\overline{y}) \stackrel{\Delta}{=} P$ where all free variables of $P$ are in the set of pairwise distinct variables $\overline{y}$. Hence, $p(\overline{x})$ evolves into $P[\overline{x}/\overline{y}]$ where each $x_i$ replaces the corresponding formal parameter $y_i$.

**Definition 2.5. (Declaration and CCP program)**
A CCP program takes the form $\mathcal{D}.Q$ where $\mathcal{D}$ is a set of process declarations of the form $p(\overline{y}) \stackrel{\Delta}{=} P$ and $Q$ is an agent.

**Operational Semantics.** The Structural Operational Semantics (SOS) of CCP is given by the transition relation $\gamma \longrightarrow \gamma'$ satisfying the rules in Figure 1. Here we follow the formulation in [27] where the local variables created by the program appear explicitly in the transition system. Processes are quotiented by a structural congruence relation $\cong$ satisfying: (STR1) $P \cong Q$ if they differ only by a renaming of bound variables ($\alpha$-conversion); (STR2) $P \parallel Q \cong Q \parallel P$; (STR3) $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$; (STR4) $P \parallel \mathbf{skip} \cong P$.

The axioms of associativity and commutativity for parallel composition make possible to give to agents a structure of multiset [27]. So, from now on with a slight abuse of notation we identify a parallel composition of agents by the corresponding multiset of agents. Thus, the sequence of processes $\Gamma = P_1, P_2, \ldots, P_n$, modulo STR2 and STR3 can be seen as a multiset, and represents the process $P_1 \parallel P_2 \parallel \cdots \parallel P_n$. We shall indistinguishably use both notations to denote parallel composition. Moreover, we may also write, e.g., $\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma$ to emphasize that the body of the **ask** agent is a parallel composition of a (possibly singleton) multiset of processes.

A *configuration* $\gamma$ is a triple of the form $(X; \Gamma; c)$, where $c$ is a constraint representing the store, $\Gamma$ is a multiset of processes, and $X$ is a set of hidden (local) variables of $c$ and $\Gamma$. Let us briefly explain the transition rules for configurations in Figure 1. A tell agent $\mathbf{tell}(c)$ adds $c$ to the current store $d$ (Rule $\mathrm{R}_{\mathrm{TELL}}$); the process $\sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i$ executes $P_k$ if its corresponding guard $c_k$ can be entailed from the store (Rule $\mathrm{R}_{\mathrm{SUM}}$); a local process $(\mathbf{local}\, x)\, P$ adds $x$ to the set of hidden variable $X$ when no clashes of variables occur (Rule $\mathrm{R}_{\mathrm{LOC}}$). Observe that Rule $\mathrm{R}_{\mathrm{EQUIV}}$ can be used to do $\alpha$-conversion if the premise of $\mathrm{R}_{\mathrm{LOC}}$ cannot be satisfied; the call $p(\overline{x})$ executes the body of the process definition (Rule $\mathrm{R}_{\mathrm{CALL}}$), so the process $p(\overline{x})$ declared by $p(\overline{y}) \stackrel{\Delta}{=} P$ evolves into $P[\overline{x}/\overline{y}]$.

**Definition 2.6. (Observables)**
Let $\longrightarrow^*$ denote the reflexive and transitive closure of $\longrightarrow$. For any constraint $c$, if $(X; \Gamma; d) \longrightarrow^* (X'; \Gamma'; d')$ and $\exists X'.d' \models c$ we write $(X; \Gamma; d) \Downarrow_c$. If $X = \emptyset$ and $d = \mathtt{t}$ we simply write $\Gamma \Downarrow_c$.

Intuitively, if $P$ is a process then $P \Downarrow_c$ says that $P$ can reach a store $d$ strong enough to entail $c$, *i.e.*, $c$ is an output of $P$. Note that the variables in $X'$ above are hidden from $d'$ since the information about them is not observable. Note that $(X; \Gamma; d) \Downarrow_c$ iff $((\mathbf{local}\, X)\, (\Gamma \parallel \mathbf{tell}(d))) \Downarrow_c$.

## 2.2. Modalities in CCP languages

Similar to other process calculi, CCP has been extended with different modalities/behaviors to reason about a wider range of phenomena and systems [3]. In this section we present some of these extensions that rely on either different views of the constraints system (case of `eccp`, `sccp` and `lcc`) or the addition of new constructs in the language of processes (case of `ntcc`).

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d)\longrightarrow(X;\mathbf{skip},\Gamma;c\sqcup d)}\ \mathrm{R_{TELL}} \qquad \frac{d\models c_k \quad k\in I}{(X;\sum_{i\in I}\mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i,\Gamma;d)\longrightarrow(X;P_k,\Gamma;d)}\ \mathrm{R_{SUM}}$$

$$\frac{x\notin X\cup fv(d)\cup fv(\Gamma)}{(X;(\mathbf{local}\,x)\,P,\Gamma;d)\longrightarrow(X\cup\{x\};P,\Gamma;d)}\ \mathrm{R_{LOC}} \qquad \frac{p(\overline{y})\triangleq P\in\mathcal{D}}{(X;p(\overline{x}),\Gamma;d)\longrightarrow(X;P[\overline{x}/\overline{y}],\Gamma;d)}\ \mathrm{R_{CALL}}$$

$$\frac{(X;\Gamma;c)\cong(X';\Gamma';c')\longrightarrow(Y';\Delta';d')\cong(Y;\Delta;d)}{(X;\Gamma;c)\longrightarrow(Y;\Delta;d)}\ \mathrm{R_{EQUIV}}$$

Figure 1: Operational semantics for CCP calculi

### 2.2.1. Linearity and constraints consumption

The store in CCP is monotonic in the sense that whenever $(X;\Gamma;d)\longrightarrow(X';\Gamma';d')$, it must be the case that $d'\models d$. In other words, agents can only increase the information in the store. In linear CCP (`lcc`), constraints are built from a fragment of multiplicative intuitionistic linear logic [26]:

$$c,d ::= a \mid \mathbf{1} \mid c\otimes d \mid \exists x.c \mid !c$$

Here, $a$ is an atomic formula, i.e., a predicate symbol in the underlying signature. The unit $\mathbf{1}$ stands for the empty store (`t`), multiplicative conjunction ($\otimes$) denotes conjunction of constraints ($\sqcup$) and the banged constraint $!c$ represents an unbound constraint (that can be produced/consumed as many times as needed). In this setting, the store is no longer monotonic as intuitively shown in the following derivation where we underline the process being reduced (recall that $c\otimes\mathbf{1}\equiv c$):

$$(\emptyset;\underline{\mathbf{tell}(c)}||\mathbf{ask}\ (c)\ \mathbf{then}\ \mathbf{tell}(d);\mathbf{1})\longrightarrow(\emptyset;\underline{\mathbf{ask}\ (c)\ \mathbf{then}\ \mathbf{tell}(d)};c)\longrightarrow(\emptyset;\underline{\mathbf{tell}(d)};\mathbf{1})\longrightarrow(\emptyset;\emptyset;d)$$

The definition of $basic(c)$ (Remark 2.1) in a linear CS becomes trickier. We may think of extending our previous definition with a new case $basic(c)=basic_!(d)$ if $c=!d$ where $basic_!(\cdot)$ is as $basic(\cdot)$ but it adds a bang in front of atomic propositions. However, with this definition, a result as the one in Remark 2.2 does not hold. To see that, let $a,b$ be atomic propositions. Note that in intuitionistic linear logic: $!(a\otimes!b)\not\equiv!a\otimes!b$; $!\exists x.a\not\equiv\exists x.!a$; and $!(a\otimes b)\not\equiv!a\otimes!b$. Hence, we have to define $basic(!c)=!c$ regardless of $c$ being an atomic constraint or not. This implies that, in the case of `lcc`, we will not be able to decompose entirely the constraints added into the store in the enriched semantics of Section 3.1. For instance, if only $c$ is relevant in $\mathbf{tell}(!(c\otimes d)\otimes e)$, our analysis will produce $\mathbf{tell}(!(c\otimes d)\otimes\bullet)$ instead of $\mathbf{tell}(!(c\otimes\bullet)\otimes\bullet)$

**Remark 2.7.** Let $c$ be a constraint and assume, by $\alpha$-conversion, that the bound variables in $c$ are all different and disjoint from the set of free variables in $c$. Let $basic(\cdot)$ be as in Remark 2.1 but changing $\sqcup$ with $\otimes$ and letting $basic(!d)=!d$. Then $c\equiv\exists bv(c).\ \bigotimes_{c'\in basic(c)} c'$.

### 2.2.2. Epistemic and spatial behaviors

In [9], *space functions* are added to the constraint system in order to deal with spatial distribution of information. For that, a finite set of agents $\mathcal{S}$ is considered together with a family of functions

$s_i : \mathcal{C} \to \mathcal{C}$ for all $i \in \mathcal{S}$. Intuitively, the constraint $s_i(c) \sqcup s_j(d)$ asserts that the local store of the agent $i$ (resp. $j$) is $c$ (resp. $d$). By choosing the right properties on such family of functions, it is possible to model epistemic (resp. spatial) behaviors and interpret, e.g., $s_i(s_j(c))$ as "agent $i$ knows that agent $j$ knows $c$" (resp. "$c$ is confined in the space $j$ inside the space $i$).

The language of processes in `eccp` and `sccp` extends Definition 2.4 with the construct $[P]_i$. In the epistemic case, the interpretation is that the information computed by $P$ will be known by agent $i$. In the spatial case, the interpretation is that $P$ runs in the space of the agent $i$. The specification of these behaviors is due to the following operational rules:

$$\frac{(X; P; d^i) \longrightarrow (X'; P'; d')}{(X; [P]_i, \Gamma; d) \longrightarrow (X'; [P']_i, \Gamma; d \sqcup s_i(d'))}\ \mathrm{R_S} \qquad \frac{(X; P; d) \longrightarrow (X'; P'; d')}{(X; [P]_i, \Gamma; d) \longrightarrow (X'; [P]_i, P', \Gamma; d')}\ \mathrm{R_E}$$

In $\mathrm{R_S}$, $d^i$ represent the information available in the space $i$ (e.g., $(s_i(c) \sqcup s_j(d))^i = c$). Note that the information produced by $P$ is confined again to the space of the agent $i$ (see $s_i(d')$ in the conclusion of the rule). In `eccp`, $s_i(c)$ not only represents that $i$ knows $c$ but also that $c$ is a fact, i.e., in an epistemic CS, we have $s_i(c) \models c$. The rule $\mathrm{R_E}$ reflects the same principle at the level of processes.

In `eccp` and `sccp`, there is no **local** operator at the level of processes nor the $\exists$ operator at the level of constraints. One of the reasons is that the notion of space ($s_i(\cdot)$ and $[\cdot]_i$) provides already a mechanism to define *local* information. Moreover, only **ask** agents, without summations, are considered. In this setting, we define $basic(c) = basic(c, \epsilon)$ where:

$$basic(c, i_1 \cdot ... \cdot i_m) = \begin{cases} s_{i_1}(...s_{i_m}(c)...) \text{ if } c \text{ is an atomic proposition, } \mathtt{t} \text{ or } \mathtt{f} \\ basic(c', i_1 \cdot ... \cdot i_m \cdot s_j) \text{ if } c = s_j(c') \\ basic(c_1, i_1 \cdot ... \cdot i_m) \cup basic(c_2, i_1 \cdot ... \cdot i_m) \text{ if } c = c_1 \sqcup c_2 \end{cases}$$

The sequence $i_1 \cdots i_m$ represents the nesting of agent-spaces and $\epsilon$ denotes no-space/modality. For instance, $basic(a \sqcup s_i(b \sqcup s_j(c))) = \{a, s_i(b), s_i(s_j(c))\}$.

**Remark 2.8.** In spatial and epistemic CSs, it is always the case that $s_i(c \sqcup d) \equiv s_i(c) \sqcup s_i(d)$ [9]. Hence, a similar result as the one in Remark 2.2 can be established for `eccp` and `sccp`.

The observable behavior in `lcc`, `eccp` and `sccp` is defined similarly as we did in Definition 2.6.

### 2.2.3. Timed CCP

Reactive systems [29] are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receives a stimulus (input) from the environment, computes an output and then waits for the next interaction with the environment. Temporal extensions of the CCP model have been proposed to specify and verify reactive systems [30, 5, 7, 6].

Here we shall focus on the `ntcc` calculus [6] (an extension of `tcc` [30]), a language that combines ideas of CCP with constructs of Synchronous Languages [29]. More precisely, time in `ntcc` is conceptually divided into *time intervals* (or *time-units*). In a particular time interval, a CCP process $P$ gets an input $c$ from the environment, it executes with this input as the initial *store*, and when it

reaches its resting point (i.e., a state from which no more computation steps are possible), it *outputs* the resulting store $d$ to the environment. The resting point determines also a residual process $Q$ that is then executed in the next time-unit. The resulting store $d$ is not automatically transferred to the next time-unit. This way, outputs of two different time-units are not supposed to be related.

The constraint system in `ntcc` is the same as in CCP. Processes are built as in Definition 2.4 without recursive calls and the following constructs are added:

$$P, Q ::= ... \mid \mathbf{next}\ P \mid \mathbf{unless}\ (c)\ \mathbf{next}\ P \mid\ !P$$

The process $\mathbf{next}\ P$ delays the execution of $P$ to the next time interval. We shall use $\mathbf{next}^n P$ to denote $P$ preceded with $n$ copies of "$\mathbf{next}$ " and $\mathbf{next}^0 P = P$. The *time-out* $\mathbf{unless}\ (c)\ \mathbf{next}\ P$ is also a unit-delay, but $P$ is executed in the next time-unit only if $c$ is not entailed by the final store at the current time interval. The replication $!P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e., unboundedly many copies of $P$ but one at a time. We note that in `ntcc`, recursive calls must be guarded by a **next** operator to avoid infinite computations during a time-unit. Then, recursive definitions can be encoded via the ! operator [31].

The operational semantics of `ntcc` considers *internal* and *observable* transitions. The internal transitions correspond to the operational steps that take place during a time-unit. The rules are the same as in Figure 1 plus:

$$\frac{d \models c}{(X; \mathbf{unless}\ (c)\ \mathbf{next}\ P, \Gamma; d) \longrightarrow (X; \Gamma; S)}\ \mathrm{R_{Un}} \qquad \frac{}{(X; !P, \Gamma; d) \longrightarrow (X; P, \mathbf{next}\ !P, \Gamma; d)}\ \mathrm{R_!}$$

The $\mathbf{unless}$ process is precluded from execution if its guard can be entailed from the current store. The process $!P$ creates a copy of $P$ in the current time-unit and it is executed in the next time-unit. The seemingly missing rule for the $\mathbf{next}$ operator is clarified below.

The *observable transition* $P \xrightarrow{(c,d)} Q$ (read as "$P$ on input $c$, reduces in one *time-unit* to $Q$ and outputs $d$") is obtained from a finite sequence of internal reductions:

$$\frac{(\emptyset; \Gamma; c) \longrightarrow^* (X; \Gamma'; d) \not\longrightarrow}{\Gamma \xrightarrow{(c, \exists X.d)} (\mathbf{local}\, X)\, F(\Gamma')}\ \mathrm{R_{Obs}} \quad \text{where } F(R) = \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{skip} \text{ or } R = \sum \mathbf{ask}\ (c_i)\ \mathbf{then}\ R'_i \\ F(R_1) \parallel F(R_2) & \text{if } R = R_1 \parallel R_2 \\ Q & \text{if } R = \mathbf{next}\ Q \text{ or } R = \mathbf{unless}\ (c)\ \mathbf{next}\ Q \end{cases}$$

The function $F(R)$ (the future of $R$) returns the processes that must be executed in the next time-unit. More precisely, it unfolds **next** and **unless** expressions. Notice that an **ask** process reduces to $\mathbf{skip}$ if its guard was not entailed by the final store. Notice also that $F$ is not defined for $\mathbf{tell}(c)$, $!Q$ or $(\mathbf{local}\, x)\, P$ processes since all of them give rise to an internal transition. Hence these processes can only appear in the continuation if they occur within a $\mathbf{next}$ or $\mathbf{unless}$ expression.

**Definition 2.9. (Observables in `ntcc`)**
The internal transition in `ntcc` is usually considered as unobservable and then, the observables in `ntcc` are defined in terms of the reflexive and transitive closure of $\Longrightarrow$. Hence, if $P = P_0 \xrightarrow{(c_1,d_1)} \cdots P_{n-1} \xrightarrow{(c_n,d_n)} P_n$ and $d_n \models d$, then we write $(P, c_1 \cdots c_n) \Downarrow_d$ read as "$P$ outputs $d$ under input $c_1 \cdots c_n$". If the input is always `t`, then we simply write $P \Downarrow_d$.

# 3. Slicing CCP-based programs

Dynamic slicing is a technique that helps the user to debug her program by simplifying a partial execution trace, thus depurating it from parts which are irrelevant to find the bug. It can also help to highlight parts of the programs which have been wrongly ignored by the execution of a wrong piece of code. Our slicing technique consists of three main steps:

**S1** *Generating a (finite) trace* of the program. For that, we propose a *enriched semantics* that generates the (meta) information needed for the slicer.

**S2** *Marking the final configuration*, to choose some of the constraints and processes that, according to the symptoms detected, should or should not be in the final configuration.

**S3** *Computing the trace slice*, to select the processes and constraints that were relevant to produce the (marked) final configuration.

The following subsections explain in detail each step.

## 3.1. Enriched Semantics (Step S1)

The slicer requires some extra information from the execution of processes. More precisely, (1) in each operational step $\gamma \longrightarrow \gamma'$, we need to highlight the process that was reduced; and (2) the constraints accumulated in the store must reflect, precisely, the contribution of each process to the store. In order to solve (1) and (2), we propose a enriched semantics that extracts the needed meta information for the slicer. The rules for the CCP calculi considered here are in Figure 2 and explained below.

The semantics considers configurations of the shape $(X; \Gamma; S)$ where $X$ is a set of hidden variables, $\Gamma$ is a sequence of processes with *identifiers* and $S$ is a set (multiset in the case of `lcc`, see Remark 2.3) of atomic constraints. Before explaining the last two components, let us introduce the following helper functions.

**Definition 3.1. (Process identifiers)**
Let $P$ be a CCP process (similarly for the extended calculi). We shall use $(P)^{\clubsuit}$ to denote the expression resulting from decorating the subterms in $P$ with unique identifiers from $\mathbb{N}$. For instance, $(\mathbf{tell}(a) \parallel \mathbf{ask}\ (c)\ \mathbf{then}\ (\mathbf{tell}(d) \parallel \mathbf{tell}(e)))^{\clubsuit} = \mathbf{tell}(a) : 0 \parallel (\mathbf{ask}\ (c)\ \mathbf{then}\ (\mathbf{tell}(d) : 2 \parallel \mathbf{tell}(e) : 3)) : 1$. As before, a sequence $\Gamma_Q = P_1 : i_1, \cdots, P_n : i_n$ denotes the parallel composition of the processes in $\Gamma_Q$. Given a sequence of decorated processes $\Gamma_Q$, we shall use $(\Gamma_Q)^{\spadesuit}$ to denote a new sequence $\Gamma'_Q$ resulting from $\Gamma_Q$ by replacing all the identifiers with new freshly generated ones. For instance, $(\mathbf{tell}(a) : 10, \mathbf{tell}(b) : 11)^{\spadesuit} = \mathbf{tell}(a) : 20, \mathbf{tell}(b) : 21$ (if 19 was the last index used).

Note that we are considering here sequences rather than multisets. As we shall see, this allows us to avoid the use of the structural congruence rules STR2 and STR3 in the enriched semantics and identify easily the reduced process in each step, thus simplifying the implementation of the framework. The context "$\Gamma, P : i, \Gamma'$" represents that $P$ is preceded and followed, respectively, by the (possibly empty) sequences of processes $\Gamma$ and $\Gamma'$. The use of indexes will allow us to distinguish, e.g., the three different occurrences of $P$ in "$\Gamma_1, P : i, \Gamma_2, P : j, (\mathbf{ask}\ (c)\ \mathbf{then}\ P : l) : k$".

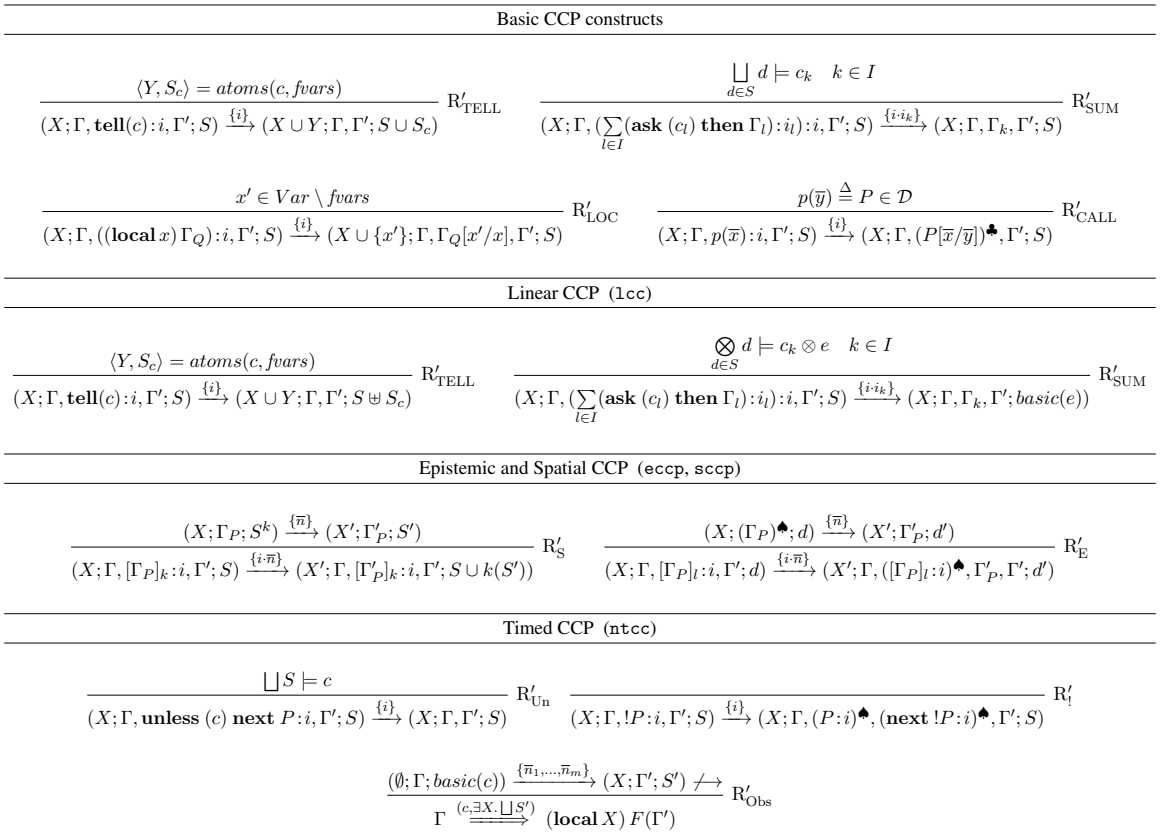


Figure 2: Enriched semantics. $\Gamma$ and $\Gamma'$ are sequences of decorated processes (Def. 3.1). In rules $\mathrm{R}'_{\mathrm{TELL}}$ and $\mathrm{R}'_{\mathrm{LOC}}$, $fvars = X \cup fv(S) \cup fv(\Gamma) \cup fv(\Gamma')$. In $\mathrm{R}'_{\mathrm{TELL}}$, $atoms(c, V)$ is in Definition 3.2, which also shows that there are no shared variables between $bv(c)$ and $fvars$. In $\mathrm{R}'_{\mathrm{S}}$, $S^k$ and $s_k(S)$ are the pointwise extensions of $d^k$ and $s_k(d)$ (see Sec. 2.2.2). In $\mathrm{R}'_{\mathrm{E}}$ and $\mathrm{R}'_{\mathrm{S}}$, $\Gamma'_P$ can be empty and then, $[]_i$ denotes $[\mathbf{skip}]_i$.In $\mathrm{R}'_{\mathrm{Obs}}$, the future function $F(\cdot)$ is the same as in `ntcc` but considering indexes.

Transitions are labeled with non-empty sequences of natural numbers locating, unequivocally the reduced process. For instance, the transition of a (decorated) process $\mathbf{tell}(c):7$ will be labeled as $\xrightarrow{\{7\}}$ (Rule $\mathrm{R}'_{\mathrm{TELL}}$); and the transition of $((\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_c):13 + (\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_d):23):8$ will be labeled as $\xrightarrow{\{8\cdot 23\}}$ denoting that the second branch was selected for execution ( Rule $\mathrm{R}'_{\mathrm{SUM}}$). If the sum has only one element, e.g., $(\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_c):13$ then the transition will have a single index ($\xrightarrow{\{13\}}$). Rule $\mathrm{R}'_{\mathrm{LOC}}$ can be explained similarly. In rule $\mathrm{R}'_{\mathrm{CALL}}$, we add the needed indexes to the body $P$ of the definition (see $(P[\overline{x}/\overline{y}])^{\clubsuit}$ in the conclusion of the rule).

**Stores and Configurations.** The solution for (2) amounts to consider the store, in a configuration, as a set/multiset of (atomic) constraints and not as a constraint. Then, the store $\{c_1, \cdots, c_n\}$ represents the constraint $c_1 \sqcup \cdots \sqcup c_n$. In order to decompose a constraint into its atomic components, we shall

use the following helper function.

**Definition 3.2. (Atomic constraints)**
Let $V \subseteq \mathit{Vars}$ and $c$ be a constraint. Assume also that the bound variables in $c$ are all distinct and not in $V$ (otherwise, by $\alpha$-conversion, we can find $c' \equiv c$ satisfying such condition). We define $atoms(c, V) = \langle bv(c), basic(c)\rangle$ where $basic(c)$ returns the atoms/tokens in $c$ (see Remark 2.1).

Observe that in Rule $\mathrm{R}'_{\mathrm{TELL}}$, the parameter $V$ of the function $atoms$ is the set of free variables occurring in the configuration. This is needed to perform $\alpha$-conversion of $c$ (which is left implicit in the definition of $basic(\cdot)$) to satisfy the condition on bound names in the definition above.

It is worth noting that we do not consider a rule for structural congruence in the enriched semantics. Such rule, in the system of Figure 1, played different roles. Axioms STR2 and STR3 provide agents with a structure of multiset (commutative and associative). As mentioned above, we consider in the enriched semantics sequences of processes with unique identifiers to highlight the process that was reduced in a transition. The sequence $\Gamma$ in Figure 2 can be of arbitrary length and then, any of the enabled processes in the sequence can be picked for execution. Axiom STR1 allowed us to perform $\alpha$-conversion on processes. This is needed in $\mathrm{R}_{\mathrm{LOC}}$ to avoid clash of variables. Note that the new Rule $\mathrm{R}'_{\mathrm{LOC}}$ internalizes such procedure by picking a fresh variable $x'$. Finally, Axiom STR4 can be used to simplify $\mathbf{skip}$ processes that can be introduced, e.g., by a $\mathrm{R}_{\mathrm{TELL}}$ transition. Observe that the enriched semantics does not add any $\mathbf{skip}$ into the configuration (see Rule $\mathrm{R}'_{\mathrm{TELL}}$).

**Example 3.3.** Consider the following toy example. Let $\mathcal{D}$ contain the process definition $A \stackrel{\Delta}{=} \mathbf{tell}(z > x + 4)$, $B \stackrel{\Delta}{=} \mathbf{tell}(z = 0)$ and $\mathcal{D}.P$ be a program where

$P = \mathbf{tell}(y < 7) \parallel \mathbf{ask}\ (x < 0)\ \mathbf{then}\ A + \mathbf{ask}\ (x \geq 0)\ \mathbf{then}\ B \parallel \mathbf{tell}(x = -3)$.

The following is a possible trace generated by the enriched semantics.

$$
\begin{array}{l}
\quad (\emptyset; \mathbf{tell}(y < 7) : 1, ((\mathbf{ask}\ (x < 0)\ \mathbf{then}\ A : 6) : 4 + (\mathbf{ask}\ (x \geq 0)\ \mathbf{then}\ B : 7) : 5) : 2, \mathbf{tell}(x = -3) : 3; \mathtt{t}) \\
\xrightarrow{\{1\}} (\emptyset; ((\mathbf{ask}\ (x < 0)\ \mathbf{then}\ A : 6) : 4 + (\mathbf{ask}\ (x \geq 0)\ \mathbf{then}\ B : 7) : 5) : 2, \mathbf{tell}(x = -3) : 3; y < 7) \\
\xrightarrow{\{3\}} (\emptyset; ((\mathbf{ask}\ (x < 0)\ \mathbf{then}\ A : 6) : 4 + (\mathbf{ask}\ (x \geq 0)\ \mathbf{then}\ B : 7) : 5) : 2; y < 7, x = -3) \\
\xrightarrow{\{2\cdot 4\}} (\emptyset; A : 6; y < 7, x = -3) \xrightarrow{\{6\}} (\emptyset; \mathbf{tell}(z > x + 4) : 8; y < 7, x = -3) \xrightarrow{\{8\}} (\emptyset; \epsilon; y < 7, x = -3, z > x + 4)
\end{array}
$$

### 3.1.1. Rules for variants of CCP-based calculi

Now we describe the rules for the other CCP calculi in Figure 2. In the case of `sccp`, we keep track of the nesting of spaces involved in the transition as the following example shows.

**Example 3.4. (`sccp` transitions)**
Consider the following derivation for the process $[[\mathbf{tell}(c)]_i]_j$ producing the constraint $s_j(s_i(c))$:

$$
\dfrac{\dfrac{(\emptyset; \mathbf{tell}(c) : 3; \mathtt{t}) \xrightarrow{\{3\}} (\emptyset; \epsilon; c)}{(\emptyset; [\mathbf{tell}(c) : 3]_i : 2; \mathtt{t}) \xrightarrow{\{2\cdot 3\}} (\emptyset; \epsilon; s_i(c))}\ \mathrm{R}'_{\mathrm{S}}}{(\emptyset; [[\mathbf{tell}(c) : 3]_i : 2]_j : 1; \mathtt{t}) \xrightarrow{\{1\cdot 2\cdot 3\}} (\emptyset; \epsilon; s_j(s_i(c)))}\ \mathrm{R}'_{\mathrm{S}}
$$

The case $\mathrm{R}'_{\mathrm{E}}$ is more interesting since the process $[P]_i$ is copied/replicated several times. Some of these copies can be relevant, from the point of view of the slicer defined below, and some others may not. For this reason, new indexes are created (via $(\cdot)^{\spadesuit}$) for each copy of $[P]_i$.

**Example 3.5.** (`eccp` **transitions)**
Consider the following derivation for the process $[[\mathbf{tell}(c)]_i]_j$:

$$(\emptyset; [[\mathbf{tell}(c):3]_i:2]_j:1; \mathtt{t}) \xrightarrow[\mathrm{R}'_{\mathrm{S}}.\mathrm{R}'_{\mathrm{E}}]{\{1\cdot2\cdot3\}} (\emptyset; [[\mathbf{tell}(c):5]_i:4 \parallel \mathbf{skip}]_j:1; s_j(c))$$
$$\xrightarrow[\mathrm{R}'_{\mathrm{E}}.\mathrm{R}'_{\mathrm{E}}]{\{1\cdot4\cdot5\}} (\emptyset; [[\mathbf{tell}(c):8]_i:7]_j:6 \parallel \mathbf{skip}; s_j(c), c) \xrightarrow[\mathrm{R}'_{\mathrm{E}}.\mathrm{R}'_{\mathrm{S}}]{\{6\cdot7\cdot8\}} (\emptyset; [[\mathbf{tell}(c):11]_i:10]_j:9 \parallel [\mathbf{skip}]_i; s_j(c), c, s_i(c))$$
$$\xrightarrow[\mathrm{R}'_{\mathrm{S}}.\mathrm{R}'_{\mathrm{S}}]{\{9\cdot10\cdot11\}} (\emptyset; [[\mathbf{skip}]_i]_j; s_j(c), c, s_i(c), s_j(s_i(c)))$$

For explanatory purposes, we have added the rules applied, bottom-up, in the derivation tree for each transition (see Example 3.4). In each case the derivation ends with an application of $\mathrm{R}'_{\mathrm{TELL}}$. We have added the **skip** processes generated by the standard semantics (Figure 1) and we have kept the empty nesting $[]_i \equiv [\mathbf{skip}]_i$ only to make more evident the reduction that took place. Note that an application of $\mathrm{R}'_{\mathrm{E}}$ involves the renaming of the indexes for the copy created. As we shall see, this will allow us to select, more accurately, the relevant processes needed to produce a given output.

The rules for `ntcc` can be explained similarly. Note that in $\mathrm{R}'_{!}$ the replicated process $P$ in $!P$ is copied into the current time unit with fresh identifiers.

Now we introduce the notion of observables for the enriched semantics and we show that it coincides with that of Definition 2.6 for the operational semantics. The cases for CCP, `lcc`, `eccp` and `sccp` are all similar. The correspondence in the case of `ntcc` follows easily from the correspondence of the internal transition (which is similar to that of CCP).

**Definition 3.6. (Observables: Enriched Semantics)**

We shall write $\gamma \xrightarrow{\{\overline{n}_1,\ldots,\overline{n}_m\}} \gamma'$ whenever $\gamma = (X_0; \Gamma_0; S_0) \xrightarrow{\{\overline{n}_1\}} \cdots \xrightarrow{\{\overline{n}_m\}} (X_m; \Gamma_m; S_m) = \gamma'$. Moreover, if $\exists X_m.\ \bigsqcup_{d \in S_m} d \models c$, then we write $\gamma \downarrow\downarrow_c$. If $X_0 = S_0 = \emptyset$, we simply write $\Gamma_0 \downarrow\downarrow_c$.

**Theorem 3.7. (Adequacy)**
For any CCP, `lcc`, `eccp` `sccp` and `ntcc` process $P$ and constraint $c$, $P \Downarrow_c$ iff $(P)^{\clubsuit} \downarrow\downarrow_c$

**Proof:**
Given a set of variables $V$, a constraint $d$ and a set of constraints $S$, let us use $\lfloor d \rfloor_V$ to denote (the resulting tuple) $atoms(d, V)$ and $\lceil S \rceil_V$ to denote the constraint $\exists V.\ \bigsqcup_{c_i \in S} c_i$. If $\langle Y, S\rangle = \lfloor d \rfloor_V$, from the definition of *atoms* and the Remarks 2.2, 2.7 and 2.8, we can show that $d \equiv \lceil S \rceil_Y$.

Moreover, let $\Gamma$ be a multiset and $\Psi$ be a sequence of decorated processes. Let us use $\lfloor \Gamma \rfloor$ to denote any sequence of processes with distinct identifiers built from the processes in $\Gamma$ and $\lceil \Psi \rceil$ to denote the multiset built from the processes in $\Psi$.

($\Rightarrow$) The proof proceeds by induction on the length of the derivation needed to perform the output $c$ in $P \Downarrow_c$ (and case analysis on the last rule applied). Consider a transition of the shape $\gamma = (X; \Gamma; d) \longrightarrow$

$(X'; \Gamma'; d')$. Let $\langle Y, S\rangle = \lfloor d \rfloor_V$ where $V = X \cup fv(\Gamma) \cup fv(d)$. By choosing the same process reduced in $\gamma$, we can show that there exist $\overline{n}$ s.t. the enriched semantics mimics the same transition as $(X \cup Y, \lfloor \Gamma \rfloor, S) \xrightarrow{\{\overline{n}\}} (X' \cup Y'; \lfloor \Gamma'' \rfloor; S')$ where $d' \cong \lceil S' \rceil_{Y'}$ and $\Gamma'' \cong \Gamma'$. This also proves the adequacy for the internal transition of `ntcc` processes. The adequacy for the `ntcc` observable transition follows easily by noting that the future function is the same in both semantics and the fact that $\exists X. \bigsqcup_{d \in basic(c)} d \equiv \exists X.c$ (see Remark 2.2).
The ($\Leftarrow$) side follows from similar arguments. □

## 3.2. Marking the final configuration (Step S2)

From the final configuration, the user must indicate the symptoms that are relevant to the slice that she wants to recompute. For that, she must select a set of constraints and/or processes that she considers relevant to identify a bug. Normally, these are elements at the end of a partial computation, and there are several strategies that one can follow to identify them.

Let us suppose that the final configuration in a partial computation is $(X; \Gamma; S)$. The symptoms that something is wrong in the program (in the sense that the user identifies some unexpected constraints or processes) may be (and not limited to) the following:

1. *Causality:* the user identifies, according to her knowledge, a subset $S' \subseteq S$ that needs to be explained (i.e., we need to identify the processes that produced $S'$).

2. *Variable Dependencies:* The user may identify a set of variables $V \subseteq fv(S)$ whose constraints need to be explored. Then, one would be interested in marking the following set of constraints $S_s = \{c \in S \mid vars(c) \cap V \neq \emptyset$ or $\exists y, c'$ such that $y \in vars(c'), y \in vars(c)$ and $c' \in S_s\}$, where the set $S_s$ considered is the the least set, w.r.t. to the ordering $\subseteq$, which is a fixpoint for the above expression.

3. *Unexpected behaviors*: there is a constraint $c$ entailed from the final store that is not expected from the intended behavior of the program. Then, one would be interested in marking the following set of constraints: $S_s = S_{min}(S, c)$ (see Remark 2.1 for the definition of $S_{min}$).

4. *Inconsistent output*: The final store should be consistent with respect to a given specification (constraint) $c$, i.e., $S$ in conjunction with $c$ must not be inconsistent. In this case, the set of constraints to be marked is $S_s = \bigcup\{S' \subseteq S \mid \bigsqcup S' \sqcup c \models \mathtt{f}$ and $S'$ is set minimal$\}$.

5. *Unexpected processes*: There may be, for instance, a call $p(\vec{t})$ in $\Gamma$ that should not be executed. Then, one would be interested in selecting $\Gamma_s \subseteq \Gamma$ in order to highlight the more relevant processes (and constraints) that lead to the execution of $p(\vec{t})$.

## 3.3. Trace Slice (Step S3)

Starting from the marking $(\Gamma_s, S_s)$, we define a backward slicing step. We identify, by means of a backward evaluation, the set of transitions that are necessary for introducing the elements in such marking. By doing that, we will eliminate information not related/relevant to $(\Gamma_s, S_s)$.

**Input:** (1) a trace $\gamma_0 \xrightarrow{\{\overline{i_1}\}} \cdots \xrightarrow{\{\overline{i_n}\}} \gamma_n$ where $\gamma_i = (X_i; \Gamma_i; S_i)$ (2) The initial marking $(S_s, \Gamma_s)$
**Output:** a sliced trace $\gamma'_0 \longrightarrow \cdots \longrightarrow \gamma'_n$

```
1 begin
2     let θ = {{•/i} | P:i ∈ Γ_n \ Γ_s} in
3     γ'_n ← (vars(S_s, Γ_s); Γ_n θ; S_s);
4     for l= n − 1 to 0 do
5         let (θ', S'_s) ← sliceProcess(θ, i_{l+1}, γ_l, γ_{l+1}, S_s) in
6         (S_s, θ) ← (S'_s, θ ∘ θ');
7         γ'_l ← (vars(S_l ∩ S_s, Γ_l θ) ; Γ_l θ ; S_l ∩ S_s)
8     end
9 end
```

**Algorithm 1:** Trace slicer for CCP-based calculi

**Notation 1. (Sliced Terms)**
We shall use the fresh constant symbol $\bullet$ to denote an "irrelevant" constraint or process. Then, for instance, "$c \sqcup \bullet$" results from a constraint $c \sqcup d$ where $d$ is irrelevant. Similarly, $\mathbf{ask}\ (c)\ \mathbf{then}\ (P \parallel \bullet) + \bullet$ results from a process of the form $\mathbf{ask}\ (c)\ \mathbf{then}\ (P \parallel Q) + \sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ P_l$ where $Q$ and the summands in $\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ P_l$ are irrelevant. We also assume that a sequence $\bullet, \ldots, \bullet$ (or $\bullet \sqcup \ldots \sqcup \bullet$) with any number ($\geq 1$) of occurrences of $\bullet$ is equivalent to a single occurrence.

**Definition 3.8. (Replacing substitution)**
A replacement is either a pair of the shape $\{T/i\}$ or $\{T/c\}$. In the first (resp. second) case, the process with identifier $i$ (resp. constraint $c$) is replaced with $T$. A replacing substitution $\theta$ is a set of replacements. In each replacing substitution $\theta$, if $t/s$ and $t'/s' \in \theta$, then $s \neq s'$. The composition of replacing substitutions $\theta_1$ and $\theta_2$, denoted as $\theta_1 \circ \theta_2$, is given by $(\theta_1 \cup \theta_2) \setminus \sigma$, where $\sigma = \{t/s \mid t/s \in \theta_2, t'/s \in \theta_1 \text{ and } t \neq t'\}$.

Algorithm 1 computes the slicing. The input is an already computed trace plus the initial marking from the user $(S_s, \Gamma_s)$. The first replacement $\theta$ (line 2) reduces to $\bullet$ all the processes not included in the marking $\Gamma_s$ and then, the last configuration (line 3) includes only the local variables, processes and constraints of interest. This algorithm is the base for all the CCP calculi including the internal transition of `ntcc`.

The new replacing substitutions are computed by the function $sliceProcess$ in Algorithm 2. Let us start with the base cases for CCP. Suppose that $\gamma \xrightarrow{\{\bar{i}\}} \psi$. We consider each kind of process. For instance, assume a $\mathrm{R}'_{\mathrm{TELL}}$ transition $\gamma = (X_l; \Gamma_1, \mathbf{tell}(c) : i, \Gamma_2; S_l) \xrightarrow{\{i\}} (X_{l+1}; \Gamma_1, \Gamma_2; S_{l+1}) = \psi$. We note that $X_l \subseteq X_{l+1}$ and $S_l \subseteq S_{l+1}$[1]. We replace the constraint $c$ with its sliced version $c'$ computed by the function $sliceConstraints$. In that function, we compute the contribution of $\mathbf{tell}(c)$ to the store, i.e., $S_c = S_{l+1} \setminus S_l$. Then, any atom $c_a \in S_c$ not in the relevant set of constraints $S_s$ is replaced by $\bullet$. By joining together the resulting atoms, and existentially quantifying the variables in $X_{l+1} \setminus X_l$ (if any), we obtain the sliced constraint $c'$. In order to further simplify the trace, if $c'$ is $\bullet$ or $\exists \overline{x}.\bullet$ then we substitute $\mathbf{tell}(c)$ with $\bullet$ (thus avoiding the "irrelevant" process $\mathbf{tell}(\bullet)$).

In a non-deterministic choice, all the precluded choices are discarded ("$+\ \bullet$"). Moreover, if the chosen alternative $Q_k$ does not contribute to the final store (i.e., $\Gamma_P \theta = \bullet$), then the whole process

[1] Note that the inclusion $S_\gamma \subseteq S_\psi$ does not hold for the non-monotonic variant `ltcc`.

$\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ P_l$ becomes $\bullet$. Note that we also modify $S_s$ (the initial marking) by adding the constraints needed to entail the guard $c_k$ (see definition of $S_{min}$ in Remark 2.1). As we shall see in the forthcoming example, this allows us to mark also the processes that contributed with constraints to entail the guard of the **ask** agent.

In $(\mathbf{local}\, x)\, Q$, the variable $x$ may be replaced to avoid a clash of names (see $\mathrm{R}'_{\mathrm{LOC}}$). The (new) created variable must be $\{x'\} = X_{l+1} \setminus X_l$. We check whether the continuation $\Gamma_P\theta$ is relevant or not to return the appropriate replacement. The case of procedure calls can be explained similarly.

```
1  Function sliceProcess(θ,i · i⃗', γ_l, γ_{l+1}, S_s)
2      let γ_l = (X_l; Γ, P:i, Γ'; S_l) and γ_{l+1} = (X_{l+1}; Γ, Γ_P, Γ'; S_{l+1}) in
3      match P with
4          case tell(c) do
5              let c' = sliceConstraints(X_l, X_{l+1}, S_l, S_{l+1}, S_s) in
6              if c' = • or c' = ∃x̄.• then return (S_s, {•/i}) else return (S_s, {tell(c')/i});
7          case Σ ask (c_l) then Q_l and i⃗' = k do
8              if Γ_P θ = • then
9                  return {(S_s, •/i)}
10             else
11                 S_s ← S_s ∪ S_min(S_l, c_k);return (S_s, {ask (c_k) then (Γ_P θ) + • / i})
12             end
13         case (local x) Q do
14             let {x'} = X_{l+1} \ X_l in
15             if Γ_P θ = • then return (S_s, {•/i}) else return (S_s, {(local x') Γ_P θ/i});
16         case p(ȳ) do
17             if Γ_P θ = • then return (S_s, {•/i}) else return (S_s,∅);
18     end
19 end
20 Function sliceConstraints(X_l, X_{l+1}, S_l, S_{l+1}, S_s)
21     let S_c = S_{l+1} \ S_l and θ = ∅ in
22     foreach c_a ∈ S_c \ S_s do θ ← θ ∘ {•/c_a} ;
23     return ∃(X_{l+1} \ X_l). ⨆ S_c θ
24 end
```

**Algorithm 2:** Slicing CCP processes and constraints.

**Example 3.9. (Ask agents and causality)**
Let $a, b, c, d, e$ be constraints without any entailment and consider the process

$R = \mathbf{ask}\ (a)\ \mathbf{then}\ \mathbf{tell}(c) \parallel \mathbf{ask}\ (c)\ \mathbf{then}\ (\mathbf{tell}(d) \parallel \mathbf{tell}(b)) \parallel \mathbf{tell}(a) \parallel \mathbf{ask}\ (e)\ \mathbf{then}\ \mathbf{skip}$

In any execution of $R$, the final store is $\{a, b, c, d\}$ and one possible trace (omitting the indexes) is:

```
[0 ; ask(a, tell(c)) || ask(c, tell(d) || tell(b)) || tell(a) || ask(e, skip) ; t] -->
[0 ; ask(a, tell(c)) || ask(c, tell(d) || tell(b)) || ask(e, skip) ; a] -->
[0 ; tell(c) || ask(c, tell(d) || tell(b)) || ask(e, skip) ; a]-->
[0 ; ask(c, tell(d) || tell(b)) || ask(e, skip) ; a,c] -->
[0 ; tell(d) || tell(b) || ask(e, skip) ; a,c] -->
[0 ; tell(d) || ask(e, skip) ; a,c,b] -->
[0 ; ask(e, skip) ; a,c,b,d]
```

If the user selects only $\{d\}$ as slicing criterion, our implementation (see Section 3.4) returns

```
[0 ; ask(a, tell(c)) || ask(c, tell(d) || *) || tell(a) || * ; t] -->
[0 ; ask(a, tell(c)) || ask(c, tell(d) || *) || * ; a] -->
[0 ; tell(c) || ask(c, tell(d) || *) || * ; *] -->
[0 ; ask(c, tell(d) || *) || * ; c,d] -->
```

```
[0 ; tell(d) || * ; d] -->
[0 ; tell(d) || * ; d] -->
[0 ; * ; d]
```

In this simplified trace, only the relevant part of the processes needed to produce $d$ is highlighted. Note that the constraints $c$ and $a$ are also added as highlighted in the trace (see $S_s \leftarrow S_{min}(S_s, c_k)$ in the case of the **ask** process in Algorithm 2). This is useful to capture the *causality* relation showing that the process $\mathbf{tell}(a)$ is needed to finally produce the constraint $d$.

### 3.3.1. A trace slicer for `lcc`

The slicing for `lcc` requires only to adapt the case for **ask** agents since those agents consume information when evolving. Note that in the case of an **ask** transitions, we have $S_{l+1} \subseteq S_l$ (see parameters in line 2 of Algorithm 2). Then, as shown in Algorithm 3, the whole **ask** agent is irrelevant if (1) $\Gamma_P\theta$ is irrelevant and (2) the transition did not consume any information of interest. In other words, an **ask** process $Q$ is highlighted either if the body of $Q$ adds relevant information (as in CCP) or if $Q$ consumes relevant information from the store. This will be useful to detect processes that may compete for the same token of information as illustrated in the example below.

1 **case** $\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ Q_l$ *and* $\vec{i} = k$ **do**
2 | **if** $\Gamma_P\theta = \bullet$ **and** $S_s \cap (S_l \setminus S_{l+1}) = \emptyset$ **then**
3 | | **return** $(S_s, \{\bullet / i\})$
4 | **else**
5 | | $S_s \leftarrow S_s \uplus basic(c_k)$;
6 | | **return** $(S_s, \{\mathbf{ask}\ (c_k)\ \mathbf{then}\ (\Gamma_P\theta) + \bullet\ /\ i\})$
7 | **end**

**Algorithm 3:** Case for ask agents in `lcc`. $\Gamma_P$ is in line 2 of Algorithm 2.

**Example 3.10. (`lcc` synchronization)**
Consider now the same atomic constraints of the previous example and the following `lcc` processes: $Q = \mathbf{ask}\ (a)\ \mathbf{then}\ \mathbf{ask}\ (b)\ \mathbf{then}\ \mathbf{tell}(c \otimes a \otimes b)$, $R = \mathbf{ask}\ (b)\ \mathbf{then}\ \mathbf{ask}\ (a)\ \mathbf{then}\ \mathbf{tell}(d \otimes a \otimes b)$ and $P = \mathbf{tell}(a \otimes b)$. A possible execution of this process is as follows:

```
[0 ; ask(a, ask(b, tell(c x a x b))) || ask(b, ask(a, tell(d x a x b))) || tell(a x b) ; 1] -->
[0 ; ask(a, ask(b, tell(c x a x b))) || ask(b, ask(a, tell(d x a x b)))  ; a,b] -->
[0 ; ask(b, tell(c x a x b)) || ask(b, ask(a, tell(d x a x b)))  ; b] -->
[0 ; ask(b, tell(c x a x b)) || ask(a, tell(d x a x b))  ; 1]
```

One may think that $Q$ consumes $a$ and $b$, then it releases those tokens and $R$ can be successfully executed. Hence, it seems reasonable to have, as an output, the constraints $c$ and $d$. Notice that this is not the case in the trace above that finishes with two blocking asks. In fact, the user may have in her mind[2] the linear logic equality $a \multimap (b \multimap (a \otimes b \otimes c)) \equiv (a \otimes b) \multimap (a \otimes b \otimes c)$ and there is no reason for the processes above to be in a deadlock. As shown in [32], the processes $\mathbf{ask}\ (a)\ \mathbf{then}\ \mathbf{ask}\ (b)\ \mathbf{then}\ P$ and $\mathbf{ask}\ (a \otimes b)\ \mathbf{then}\ P$ are logically equivalent in the sense of [27] but they have different concurrent behaviors. By selecting the two blocking **ask** agents as slicing criterion $\Gamma_s$, the simplified trace makes evident the two competing agents:

[2] Ask agents can be seen as linear implications ($\multimap$) as shown in [27].

```
[0 ; ask(a, ask(b, *)) || ask(b, ask(a, *)) || tell(a x b) ; 1] -->
[0 ; ask(a, ask(b, *)) || ask(b, ask(a, *))  ; a,b] -->
[0 ; ask(b, *) || ask(b, ask(a, *))  ; b] -->
[0 ; ask(b, *) || ask(a, *)  ; 1]
```

The reader may argue that the resulting trace is not much different from the original one. We note that the synchronization mechanism used in this example is exactly the same used in [27] to model the Dining Philosophers (DP) problem ($a, b$ are the forks –that must be grabbed in one atomic step– and $c, d$ the action of "eating"). In the DP problem, $P, Q, R$ above run in parallel with many other processes that will be simplified to $\bullet$. More interestingly, in [33], the synchronization of `lcc` processes is used to model the flow of access permissions in a concurrent object oriented (OO) programming language. The encoding of a simple OO program may generate more than 1000 `lcc` concurrent processes and then, the simplification above can definitely play an important role in finding possible bugs.

### 3.3.2. A trace slicer for `eccp` and `sccp`

The new cases for the `eccp` and `sccp` calculi are in Algorithm 4. The first case corresponds to an application of the rule $\mathrm{R}'_{\mathrm{S}}$ and we distinguish two subcases depending whether the reduced process is an **ask** or a **tell**. In the **ask** case, notice that we do not add to $S_s$ (line 7) the constraints entailing $c$ but the set of constraints entailing a constraint of the form $s_{a_1}(...(s_{a_m}(c)))$ where the $a_i$ are the nested spaces where the process is located. This is the purpose of the function $outer$ (see Caption in Algorithm 4). The **tell** case is similar to the previous cases. Note the use of the function $inner$ (line 10) to deal with the nesting of processes.

In $\mathrm{R}'_{\mathrm{E}}$, as we illustrate in the following example, the renaming of indexes guarantees that the replacement computed will never apply to different copies of the same process.

**Example 3.11. (Local stores)**
Consider the trace in Example 3.5 that we repeat here for the sake of clarity:

$$(\emptyset; [[\mathbf{tell}(c):3]_i:2]_j:1; \mathtt{t}) \xrightarrow[\mathrm{R}'_{\mathrm{S}}.\mathrm{R}'_{\mathrm{E}}]{\{1\cdot2\cdot3\}} (\emptyset; [[\mathbf{tell}(c):5]_i:4 \parallel \mathbf{skip}]_j:1; s_j(c)) \xrightarrow[\mathrm{R}'_{\mathrm{E}}.\mathrm{R}'_{\mathrm{E}}]{\{1\cdot4\cdot5\}} (\emptyset; [[\mathbf{tell}(c):8]_i:7]_j:6 \parallel \mathbf{skip}; s_j(c), c)$$
$$\xrightarrow[\mathrm{R}'_{\mathrm{E}}.\mathrm{R}'_{\mathrm{S}}]{\{6\cdot7\cdot8\}} (\emptyset; [[\mathbf{tell}(c):11]_i:10]_j:9 \parallel [\mathbf{skip}]_i; s_j(c), c, s_i(c)) \xrightarrow[\mathrm{R}'_{\mathrm{S}}.\mathrm{R}'_{\mathrm{S}}]{\{9\cdot10\cdot11\}} (\emptyset; [[\mathbf{skip}]_i]_j; s_j(c), c, s_i(c), s_j(s_i(c))))$$

Assume now that the user is only interested in the constraints under the scope of the agent $j$, i.e., she selects $S_s = \{s_j(c), s_j(s_i(c))\}$. The slicer will report the following trace. In order to clarify the result, we also explicitly show the replacements computed at each step.

```
( 0 ; [ [ tell(c)] i ]j ; t )    -->              *** { tell(c) / 3 }, i.e., no changes
( 0 ; [ [ * ]i ]j        ; sj(c) ) -->            *** { * / 5 } but 4 remains (due to the copy 6)
( 0 ; [ [ * ]i ]j        ; sj(c) ) -->            *** { * / 8 } but 6 remains (due to the copy 9)
( 0 ; [ [tell(c)]i ]j    ; sj(c))  -->            *** { tell(c) / 11 }, i.e., no changes
( 0 ; *                  ; sj(c), sj(si(c)) )     *** initial marking containing only j-constraints
```

Note that the processes indexed with 4 and 6 are not replaced with $\bullet$ since the copy (due to rule $\mathrm{R}'_{\mathrm{E}}$) indexed with 9 contributed with the constraint $s_j(s_i(c))$ in the last transition. A post-processing of the sliced trace could do an extra simplification to

```
( 0 ; [ [ tell(c)] i ] j ; t )  --> ( 0 ; [ [tell(c)]i ]j ; j(c)) --> ( 0 ; * ; j(c), j(i(c)) )
```

by substituting $[[\bullet]_i]_j$ with $\bullet$. This trace clearly shows the two transitions that added $j$-constraints.

1 **case** $[\Gamma_Q]_a$ **and** $\Gamma_P = [\Gamma'_Q]_a$ *# Case for a* $\mathrm{R}'_{\mathrm{S}}$ *(spatial) transition* **do**
2 | **match** $R$ **with**
3 | | **case** $\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_R$ **do**
4 | | | **if** $\Gamma_R\theta = \bullet$ **then**
5 | | | | $\theta' \leftarrow \{\bullet/i_n\}$
6 | | | **else**
7 | | | | $\theta' \leftarrow \{\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_R\theta/i_n\}; S_s \leftarrow S_s \cup S_{min}(S_l, outer(i_1 \cdots i_{n-1}, c))$
8 | | | **end**
9 | | **case** $\mathbf{tell}(c)$ **do**
10 | | | **let** $c' = inner(i_1 \cdots i_{n-1}, sliceConstraints(X_l, X_{l+1}, S_l, S_{l+1}, S_s))$ **in**
11 | | | **if** $c' = \bullet$ **then** $\theta' \leftarrow \{\bullet/i_n\}$ **else** $\theta' \leftarrow \{\mathbf{tell}(c')/i_n\}$;
12 | | **if** $\Gamma_P(\theta \circ \theta') = \bullet$ **then return** $(S_s, \theta' \circ \{\bullet/i_1\})$ **else return** $(S_s, \theta')$;
13 | **end**
14 **case** $[\Gamma_Q]_a$ **and** $\Gamma_P = [\Gamma_Q]_a, \Gamma'_Q$ *# Case for a* $\mathrm{R}'_{\mathrm{E}}$ *(epistemic) trantion* **do**
15 | **match** $R$ **with**
16 | | **case** $\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_R$ **do**
17 | | | **if** $\Gamma_R\theta = \bullet$ **then**
18 | | | | $\theta' \leftarrow \{\bullet/i_n\}$
19 | | | **else**
20 | | | | $\theta' \leftarrow \{\mathbf{ask}\ (c)\ \mathbf{then}\ \Gamma_R\theta/i_n\}; S_s \leftarrow S_s \cup S_{min}(S_l, outer(i_1 \cdots i_{n-1}, c))$
21 | | | **end**
22 | | **case** $\mathbf{tell}(c)$ **do**
23 | | | **let** $c' = inner(i_1 \cdots i_{n-1}, sliceConstraints(X_l, X_{l+1}, S_l, S_{l+1}, S_s))$ **in**
24 | | | **if** $c' = \bullet$ **then** $\theta' \leftarrow \{\bullet/i\}$ **else** $\theta' \leftarrow \{\mathbf{tell}(c')/i_n\}$;
25 | | **if** $\Gamma_P(\theta \circ \theta') = \bullet$ **then return** $(S_s, \theta' \circ \{\bullet/i_1\})$ **else return** $(S_s, \theta')$;
26 | **end**

**Algorithm 4:** Cases for `eccp` and `sccp`. Let the parameter $i.\vec{i'}$ in $sliceProcess$ (Algorithm 2) be the sequence $i_1 \cdots i_n$ of indexes. $R$ in line 2 and 15 is the process labeled with $i_n$. We decree that $[\bullet]_a = s_i(\bullet) = \bullet$. We define $outer(\epsilon, c) = c$; and $outer(i.\overline{n}, c) = outer(\overline{n}, s_{a_i}(c))$ whenever $i$ is an index of a process of the form $[P]_{a_i}$. We also define $inner(i_1 \cdots i_m, s_{a_{i_1}}(...s_{a_{i_m}}(c))) = c$.

### 3.3.3. A trace Slicer for `ntcc`

From the execution point of view, only the `ntcc` observable transition is relevant since it describes the input-output behavior of processes. However, when a `ntcc` program is debugged, we have to consider also the internal transitions. The cases for the slicer are in Algorithm 5. If an **unless** process evolves (into **skip**) during a time-unit, then it is irrelevant. In the case of $!\Gamma_P$, we note that $\Gamma_Q$ contains a freshly created copy of $\Gamma_P$ plus a process of the form $\mathbf{next}\ !\Gamma_P$ also with new indexes. Similar to the case of `eccp`, we check whether at least one of those processes contributed to the final store. Otherwise, that particular copy of $!\Gamma_P$ is reduced to $\bullet$.

```
1 case unless (c) next Γ'_P do
2   |  return (S_s, [•/i])
3 end
4 case !Γ'_P do
5   |  if Γ_P θ = • then return (S_s, {•/i}) else return (S_s, {!(Γ_P θ)/i});
6 end
```

**Algorithm 5:** Cases for `ntcc`

Recall that **next** processes do not exhibit any transition during a time-unit and then, this case does not need to be considered in the algorithm.

For the observable transition we proceed as follows. Consider a trace of $n$ observable steps $\gamma_0 \Longrightarrow \cdots \Longrightarrow \gamma_n$ and the marking $(S_s, \Gamma_s)$ to be highlighted in the last configuration $\gamma_n$. Let $\theta_n$ be the replacement computed during the slicing process of the (internal) trace generated from $\gamma_n$. We propagate the replacements in $\theta_n$ to the configuration $\gamma_{n-1}$ as follows:

1. In $\gamma_{n-1}$ we set $S_s = \emptyset$. Note that the unique store of interest for the user is the one in $\gamma_n$. Recall also that the final store in `ntcc` is not transferred to the next time-unit. Then, only the processes (and not the constraints) in $\gamma_{n-1}$ are responsible for the final store in $\gamma_n$.

2. Let $\psi$ be the last internal configuration in $\gamma_{n-1}$, i.e., $\gamma_{n-1} \xrightarrow{\{\bar{i}_1,\ldots,\bar{i}_m\}} \psi \not\longrightarrow$ and $\gamma_n = F(\psi)$. We propagate the replacements in $\theta_n$ to $\psi$ before running the slicer on the trace starting from $\gamma_{n-1}$. For that, we compute a replacement $\theta'$ that must be applied to $\psi$ as follows:

   - If there is a process $R = \mathbf{next}\ P : i$ in $\psi$, then $\theta'$ includes the replacement $\{\mathbf{next}\ (\Gamma_P\theta_n)/i\}$. For instance, if $R = \mathbf{next}\ (\mathbf{tell}(c) \parallel \mathbf{tell}(d))$ and $\mathbf{tell}(c)$ was irrelevant in $\gamma_n$, the resulting process in $\psi$ is $\mathbf{next}\ (\bullet \parallel \mathbf{tell}(d))$ . The case for $\mathbf{unless}\ (c)\ \mathbf{next}\ P$ is similar.
   - If there is a process $R = \sum_l \mathbf{ask}\ (c_l)\ \mathbf{then}\ P_l : i$ in $\psi$ (which is irrelevant since it was not executed), we add to $\theta'$ the replacement $\{\bullet/i\}$.

3. Starting from $\psi\theta$, we compute the slicing on $\gamma_{n-1}$ (Algorithm 1).

4. This procedure continues until the first configuration $\gamma_0$ is reached.

**Example 3.12.** Consider the following process definitions:

$System \triangleq Beat2 \parallel Beat4 \qquad Beat2 \triangleq \mathbf{tell}(b2) \parallel \mathbf{next}^{\,2}\ Beat2$

$Beat4 \triangleq \mathbf{tell}(b4) \parallel \mathbf{next}^{\,4}\ Beat4$

This is a simple model of a multimedia system that, every 2 (resp. 4) time-units, produces the constraint $b2$ (resp. $b4$). Then, every 4 time-units, the system produces both $b2$ and $b4$. If we compute 5 time-units and choose $S_s = \{b4\}$ we obtain:

```
{1 / 5 > [0 ; System ; *] -->  [0 ; Beat4 ; *] --> [0 ; next^4(Beat4) ; *]} ==>
{2 / 5 > [0 ; next^3(Beat4) ; *]} ==>
{3 / 5 > [0 ; next^2(Beat4) ; *] } ==>
{4 / 5 > [0 ; next(Beat4) ; *]} ==>
{5 / 5 > [0 ; Beat4 ; *] --> [0 ; tell(b4) || * ; *] --> [0 ; * ; b4]}
```

Note that all the executions of $Beat2$ in time-units 1, 3 and 5 are hidden since they do not contribute to the observed output $b4$. More interestingly, the execution of $\mathbf{tell}(b4)$ in time-unit 1, as well as the recursive call of $Beat4$ ($\mathbf{next}^{\,4}\ Beat4$) in time-unit 5, are also hidden.

Now assume that we compute an even number of time-units. Then, no constraint is produced in that time-unit and the whole execution of $System$ is hidden:

```
{1/4 > [0 ; * ; *]} ==> {0 ; 2/4 > [* ; *]} ==>
{3/4 > [0 ; * ; *]} ==> {0 ; 4/4 > [* ; *]}
```

### 3.4. Slicing tools and more experiments

We implemented in Maude[3] a prototypical version of a slicer described here that can be found at `http://subsell.logic.at/slicer/`.

In that webpage the reader can find two compelling applications of our techniques: (1) an extended version of Example 3.12 where a rhythmic pattern of a Central African Republic music [34] was programmed in `ntcc` [35]. Our slicer was able to highlight a synchronization problem between several $Beat$ processes that have to add the constraint `beat` at different time-units. (2) We specified a biochemical system where constraints of the form `Mdm2` (resp. `Mdm2A`) state that the protein `Mdm2` is present (resp. absent). The model includes activation and inhibition of biological rules modeled as processes (omitting some details) of the form $\mathbf{ask}\ (\texttt{Mdm2A})\ \mathbf{then\ next\ tell}(\texttt{Mdm2})$ modeling that "if `Mdm2` is absent now, then it must be present in the next time-unit". It may be also the case that neither `Mdm2A` nor `Mdm2` can be entailed in a given time-unit, representing the fact that we only have partial information on the system (and the current state of `Mdm2` is unknown). The interaction of many activation/inhibition rules makes the model trickier since rules may compete for resources and then, we can wrongly observe at the same time-unit that `Mdm2` is both present and absent. The concrete trace is quite obfuscated and difficult to understand. The slicer was able to simplify such trace, making it easy to find the problem in the specification.

We are currently adapting the `lcc` interpreter in [33] and implementing on top of it the Algorithm 3. This implementation encodes concurrent Java-like programs as `lcc` processes in order to perform analyses such as deadlock detection. The generated `lcc` program may include thousands of concurrent processes. As shown in `http://subsell.logic.at/alcove2/`, the traces of such program are very difficult to understand. Our techniques certainly may help to highlight the processes of interest on it.

DSpaceNet[4] is a tool for social networking based on a multi-agent spatial and timed concurrent constraint language [36]. The fundamental structure of DSpaceNet is that of spaces that may con-

[3] `http://maude.cs.illinois.edu`

[4] `http://www.dspacenet.com/`

tain spatial-mobile-reactive ntcc programs, and other spaces. A space of an agent $j$ within the space of agent $i$ means that agent $i$ allows agent $j$ to use a computation sub-space within its space. The constraint system allows users to specify advanced text message deduction, arithmetic deductions, scheduling, etc. The epistemic interpretation of spaces (eccp) can be used to derive whether there are users with conflicting/inconsistent information, or whether a group of agents may be able to deduce certain messages. DSpaceNet is intended to be an academic platform with interesting application in learning concurrent programming languages. In fact, the users are able to write ntcc-like programs among different spaces and test the result of the computation. In this learning phase, mistakes in programs (and unexpected behaviors) are common place. The techniques developed here apply straightforwardly to DSpaceNet programs. We thus plan to develop a plugin for DSpaceNet for slicing user's programs.

## 3.5. Soundness

We conclude here by showing that the slicing procedure computes a suitable approximation of the concrete trace. Our notion of approximation roughly says that a process $P'$ approximates $P$, notation $P \preceq^\sharp P'$, if $P'$ is as $P$ but replacing some subterms with $\bullet$ via a (possibly empty) replacement substitution. More formally,

**Definition 3.13. (Approximation)**
Given two constraints $c, d$, we write $c \preceq^\sharp d$ if there exists $c' \equiv c$ and replacing substitution $\theta$ s.t. $d = c'\theta$. We extend this notion to CCP processes by considering the least reflexive relation $\preceq^\sharp$ on processes satisfying

(0) $P \preceq^\sharp \bullet$ for any $P$;
(1) $\mathbf{tell}(c) \preceq^\sharp \mathbf{tell}(c')$ whenever $c \preceq^\sharp c'$ ;
(2) $\sum_{i\in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i \preceq^\sharp \mathbf{ask}\ (c_k)\ \mathbf{then}\ P'_k + \bullet$ if $k \in I$ and $P_k \preceq^\sharp P'_k$ ;
(3) $(\mathbf{local}\, x)\ P \preceq^\sharp (\mathbf{local}\, x')\ P'$ if either $x = x'$ and $P \preceq^\sharp P'$ or $x'$ does not occur free in $P$ and $P[x'/x] \preceq^\sharp P'$.

We lift the notion of approximation to sets of parallel (indexed) processes and configurations by decreeing that $(X;\Gamma;S) \preceq^\sharp (X';\Gamma';S')$ if $X' \subseteq X$, $S' \setminus \{\bullet\} \subseteq S$ and for any (indexed) process $P : i \in \Gamma$, either $P' : i \in \Gamma'$ and $P \preceq^\sharp P'$ or there is no process in $\Gamma'$ indexed with $i$.

Note that $\preceq^\sharp$ is reflexive and then for any process $P$, $P \preceq^\sharp P$ (meaning that all the subterms of $P$ are relevant). Moreover, by definition, $P \preceq^\sharp \bullet$ (and $P$ is completely irrelevant for the final output). The other cases in the definition of $\preceq^\sharp$ mirror the replacements produced by the slicing algorithm. For instance, a process $\mathbf{tell}(c)$ decomposes $c$ into its atoms (Rule $\mathrm{R}'_{\mathrm{TELL}}$) and then, some of these atoms are abstracted and joined back to form an (abstracted) constraint (function $sliceConstraint$). Hence, what we get is an equivalent constraint $c'$ where the order of the atoms may have changed, some renaming of local variables may have occurred (function $atoms$ in Rule $\mathrm{R}'_{\mathrm{TELL}}$) and some of the atoms were replaced with $\bullet$. The reader may compare the other cases in Definition 3.13 with the corresponding cases in the $sliceProcess$ function. We finally note that the abstraction procedure may eliminate some variables and also some (non-relevant) atomic constraints (see line 6 in Algorithm 1). This justifies our definition of $\preceq^\sharp$ on configurations.

The approximation relation $\preceq^\sharp$ can be extended to the variants of CCP considered here as follows. For `eccp` and `sccp`: (4) $[P]_i \preceq^\sharp [P']_i$ whenever $P \preceq^\sharp P'$; for `ntcc`, whenever $P \preceq^\sharp P'$: (5) $\mathbf{next}\ P \preceq^\sharp \mathbf{next}\ P'$, $\mathbf{unless}\ (c)\ \mathbf{next}\ P \preceq^\sharp \mathbf{unless}\ (c)\ \mathbf{next}\ P'$ and $!\,P \preceq^\sharp !\,P'$.

We can now prove that our approximation is safe. Namely, the resulting sliced trace is a correct approximation and it contains the atomic constraints needed to correctly synchronize the guards of the relevant ask processes.

**Theorem 3.14. (Approximation)**
The following holds:

(a) Let $\gamma_0 \xrightarrow{\{\bar{i}_1\}} \cdots \xrightarrow{\{\bar{i}_n\}} \gamma_n$ be a partial computation in (one of) CCP, `lcc`, `eccp sccp` and `ntcc` and $\gamma'_0 \xrightarrow{\{\bar{i}_1\}} \cdots \xrightarrow{\{\bar{i}_n\}} \gamma'_n$ be the resulting sliced trace according to an arbitrary slicing criterion. Then, for all $t \in 0..n$, $\gamma_t \preceq^\sharp \gamma'_t$; and

(b) Assume that $\gamma_{t-1} = (X_{t-1}; \Gamma, Q : i, \Gamma'; S_{t-1}) \xrightarrow{\{i \cdot k\}} (X_t; \Gamma, Q' : k, \Gamma'; S_t) = \gamma_t$ and $Q = \sum \mathbf{ask}\ (c_j)\ \mathbf{then}\ P_j$, for $t \in 1..n$. If $Q'$ is not abstracted with $\bullet$ in $\gamma'_t$ then $\bigsqcup S'_{t-1} \models c_k$.

**Proof:**
We present the proof for the case of CCP. The cases for `lcc`, `eccp`/`sccp` and `ntcc` are similar, by considering the modified cases given by Algorithms 3, 4, and 5, respectively.

We prove both cases (a) and (b) simultaneously. The proof is by induction on the length $m$ of the backward computation.

Base case ($m = 0$): If $m = 0$ then there are no transitions and $\gamma_0 \not\longrightarrow$. Then the result follows by noticing that $\gamma'_0$ is as $\gamma_0$ but replacing some constraints as well as some (indexed) processes by $\bullet$ (see lines 2 and 3 and 6 in Algorithm 1).

Inductive case ($m > 0$): We assume that $\gamma_k ... \gamma_n$ (with $n - k = m$) satisfy the property and analyze the extension when $\gamma_{k-1} \xrightarrow{\{\bar{i}_k\}} \gamma_k$ by case analysis of the rule applied in such transition. Let $\gamma_{k-1} = (X_{k-1}; \Gamma, P : i, \Gamma'; S_{k-1})$ and $\gamma_k = (X_k; \Gamma, \Gamma_Q, \Gamma'; S_k)$. By inductive hypothesis we have that $\gamma_k \preceq^\sharp \gamma'_k$. Let $\gamma'_{k-1}$ be $(X'_{k-1}; \bar{\Gamma}, P', \bar{\Gamma}'; S'_{k-1})$. Then, by line 6 in Algorithm 1 we get that $X'_{k-1} \subseteq X_{k-1}$, and $S'_{k-1} \subseteq S_{k-1}$. Thus, for the required property to hold we have to prove that $\Gamma, P : i, \Gamma' \preceq^\sharp \Gamma, P', \Gamma'$. By induction we know that $\gamma_k \preceq^\sharp \gamma'_k$ and we have to prove that $P \preceq^\sharp P'$. Let us consider the different replacing substitution items returned by the function $sliceProcess$ according to the case of the process $P$.

- Case $\mathbf{tell}(c)$. By line 6 of Algorithm 2 we have the following possibilities. (i) $\{\bullet/i\}$ is returned, and hence the new accumulated replacement substitution will replace $\mathbf{tell}(c)$ by $\bullet$ when applied to $\gamma_{k-1}$, to produce $P' = \bullet$, and clearly $P \preceq^\sharp \bullet = P'$. (ii) $\mathbf{tell}(c')/i$ is returned, where $c'$ is the constraint $c$, where some of its atomic components get replaced by $\bullet$ (and hence, $c \preceq^\sharp c'$). Then, $P = tell(c) \preceq^\sharp tell(c') = P'$.
- Case $\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ Q_l$. Here we have two subcases. Let $i = i'.t$ (i) If $Q_t\theta = \bullet$, which means that the reduction of this guarded process does not contribute to the relevant processes, then the

algorithm returns $\{\bullet/i'\}$ which means that the entire choice process gets replaced by $\bullet$ in $\Gamma'_{k-1}$. This trivially satisfies properties (a) and (b). (ii) If $Q_l\theta \neq \bullet$ then by line 8 of Algorithm 2, we get $S_s \leftarrow S_s \cup S_{min}(S_s, c_t)$ and the algorithm returns $\{\mathbf{ask}\ (c_t)\ \mathbf{then}\ (Q_t\theta) + \bullet\ /\ i'\}$. Thus, $c_t$ gets entailed in $\gamma'_{k-1}$ by definition of $S_{min}(S_s, c_t)$, and (b) holds. We can also remark that all alternatives but the one guarded by $c_t$ gets eliminated in the choice in $\gamma'_{k-1}$. We conclude (a) by noticing that, by induction, $Q_l \preceq^\sharp Q'_l = Q_l\theta$ for the replacement substitution $\theta$ computed in line 5 of Algorithm 1, and by case (2) of Definition 3.13.

- Case $(\mathbf{local}\, x)\, Q$. Here we have two possibilities. (i) $\Gamma_P\theta = \bullet$, and hence $\Gamma_P$ was considered irrelevant and $(\mathbf{local}\, x)\, Q$ is also irrelevant, gets replaced by $\bullet$ and (a) trivially holds. (ii) we simplify $(\mathbf{local}\, x)\, Q$ by replacing it with $\{(\mathbf{local}\, x')\, \Gamma_P\theta/i\}$. By definition of $\preceq^\sharp$, and since $x \neq x'$ does not appear in $\Gamma_P$, we conclude that $\{(\mathbf{local}\, x)\, \Gamma_P\} \preceq^\sharp \{(\mathbf{local}\, x')\, \Gamma_P\theta\}$ as needed.

- Case $p(\overline{y})$. Here we have two possibilities. (i) $\Gamma_P\theta = \bullet$, and hence $\Gamma_P$ was considered irrelevant and $p(\overline{y})$ gets also irrelevant and replaced by $\bullet$. Hence, the property trivially holds. (ii) $\Gamma_P$ is not irrelevant, and hence $p(\overline{y})$ remains as it is, and hence clearly approximates itself.

□

# 4. Conclusions and future work

In this paper we introduced the first framework for dynamic slicing of concurrent constraint based programs, and showed its applicability for CCP and other variants of CCP such as linear CCP [27], spatial and epistemic CCP [9] as well as with temporal extensions of it [6]. We defined the operational semantics of these languages, then we built an enriched semantics containing the technical details necessary for building a backward dynamic slicer. The user marks the relevant information in the last configuration of a partial computation and the dynamic slicer identifies and eliminates automatically the information in the trace which is not related to the marked one.

We implemented a prototype of the slicer for CCP and timed CCP in Maude and showed its use in debugging a program specifying a biochemical system and a multimedia interacting system.

We are currently working on extending our tool to cope with the other considered languages, and also to extend it and make it more friendly. We already incorporated into our framework an assertion language based on a suitable fragment of temporal logic [37]. Assertions specify invariants the program must satisfy during its execution. If an assertion is not satisfied in a given state, then the execution is interrupted and a concrete trace is generated and sliced. This was very useful to automatically detect bugs in the aforementioned biological and multimedia systems.

As future work we envisage several possible extensions of our framework, considering non deterministic constraint logic languages [38, 37], defining a forward slicer, and studying the relationship with other techniques for declarative debugging [39]. We also plan to extend our framework by studying the correlation with static analysis techniques [40, 41, 42] and develop more examples in the field of bioinformatics [43, 44, 45, 46].

**Acknowledgments**. We thank the anonymous reviewers for their detailed and very useful criticisms

and detailed recommendations that helped us to improve our paper. The work of Olarte was funded by CNPq and CAPES (Brazil). The work of Palamidessi and Olarte was supported by the Regional Program STIC AMSUD "EPIC: EPistemic Interactive Concurrency" and the project CLASSIC.

# References

[1] Saraswat VA. Concurrent Constraint Programming. MIT Press, 1993.

[2] Saraswat VA, Rinard MC, Panangaden P. Semantic Foundations of Concurrent Constraint Programming. In: Wise DS (ed.), POPL. ACM Press. ISBN 0-89791-419-8, 1991 pp. 333–352.

[3] Olarte C, Rueda C, Valencia FD. Models and emerging trends of concurrent constraint programming. *Constraints*, 2013. **18**(4):535–578.

[4] Bortolussi L, Policriti A. Modeling Biological Systems in Stochastic Concurrent Constraint Programming. *Constraints*, 2008. **13**(1-2):66–90.

[5] Saraswat VA, Jagadeesan R, Gupta V. Timed Default Concurrent Constraint Programming. *J. Symb. Comput.*, 1996. **22**(5/6):475–520. doi:10.1006/jsco.1996.0064.

[6] Nielsen M, Palamidessi C, Valencia FD. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nord. J. Comput.*, 2002. **9**(1):145–188.

[7] de Boer FS, Gabbrielli M, Meo MC. A Timed Concurrent Constraint Language. *Inf. Comput.*, 2000. **161**(1):45–83.

[8] Olarte C, Valencia FD. Universal concurrent constraint programing: symbolic semantics and applications to security. In: Wainwright RL, Haddad H (eds.), SAC. ACM. ISBN 978-1-59593-753-7, 2008 pp. 145–150.

[9] Knight S, Palamidessi C, Panangaden P, Valencia FD. Spatial and Epistemic Modalities in Constraint-Based Process Calculi. In: Koutny M, Ulidowski I (eds.), CONCUR, volume 7454 of *LNCS*. Springer. ISBN 978-3-642-32939-5, 2012 pp. 317–332.

[10] Olarte C, Pimentel E, Nigam V. Subexponential concurrent constraint programming. *Theor. Comput. Sci.*, 2015. **606**:98–120. doi:10.1016/j.tcs.2015.06.031.

[11] Codish M, Falaschi M, Marriott K. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 1994. **16**(3):649–686.

[12] Comini M, Titolo L, Villanueva A. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 2011. **11**(4-5):487–502.

[13] Falaschi M, Olarte C, Palamidessi C. Abstract interpretation of temporal concurrent constraint programs. *TPLP*, 2015. **15**(3):312–357. doi:10.1017/S1471068413000641.

[14] Shapiro EY. Algorithmic Program DeBugging. MIT Press, 1983.

[15] Weiser M. Program slicing. *IEEE Trans. on Software Engineering*, 1984. **10**(4):352–357.

[16] Korel B, Laski J. Dynamic Program Slicing. *Inf. Process. Lett.*, 1988. **29**(3):155–163. doi:10.1016/0020-0190(88)90054-3.

[17] Ochoa C, Silva J, Vidal G. Dynamic Slicing of Lazy Functional Programs Based on Redex Trails. *Higher Order Symbol. Comput.*, 2008. **21**(1-2):147–192. doi:10.1007/s10990-008-9023-7.

[18] Alpuente M, Ballis D, Espert J, Romero D. Backward Trace Slicing for Rewriting Logic Theories. In: Proc. of CADE'11. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-22437-9, 2011 pp. 34–48.

[19] Alpuente M, Ballis D, Frechina F, Romero D. Using Conditional Trace Slicing for Improving Maude Programs. *Sci. Comput. Program.*, 2014. **80**:385–415. doi:10.1016/j.scico.2013.09.018.

[20] Josep S. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.*, 2012. **44**(3):12:1–12:41.

[21] Falaschi M, Gabbrielli M, Olarte C, Palamidessi C. Slicing Concurrent Constraint Programs. In: Hermenegildo MV, López-García P (eds.), Proc. of LOPSTR 2016, volume 10184 of *LNCS*. Springer, 2017 pp. 76–93.

[22] de Boer FS, Pierro AD, Palamidessi C. Nondeterminism and Infinite Computations in Constraint Programming. *Theoretical Computer Science*, 1995. **151**(1):37–78.

[23] Hentenryck PV, Saraswat VA, Deville Y. Design, Implementation, and Evaluation of the Constraint Language cc(FD). *J. Log. Program.*, 1998. **37**(1-3):139–164.

[24] Saraswat VA. The Category of Constraint Systems is Cartesian-Closed. In: Proceedings of LICS'92. IEEE Computer Society. ISBN 0-8186-2735-2, 1992 pp. 341–345. doi:10.1109/LICS.1992.185546.

[25] Smolka G. A Foundation for Higher-order Concurrent Constraint Programming. In: Jouannaud J (ed.), CCL, volume 845 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-58403-X, 1994 pp. 50–72.

[26] Girard J. Linear Logic. *Theor. Comput. Sci.*, 1987. **50**:1–102.

[27] Fages F, Ruet P, Soliman S. Linear Concurrent Constraint Programming: Operational and Phase Semantics. *Inf. Comput.*, 2001. **165**(1):14–41.

[28] Ruet P, Fages F. Concurrent Constraint Programming and Non-commutative Logic. In: Nielsen M, Thomas W (eds.), Proceedings of CSL'97, volume 1414 of *LNCS*. Springer. ISBN 3-540-64570-5, 1997 pp. 406–423. doi:10.1007/BFb0028028.

[29] Berry G, Gonthier G. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 1992. **19**(2):87–152.

[30] Saraswat VA, Jagadeesan R, Gupta V. Foundations of Timed Concurrent Constraint Programming. In: Proceedings of LICS'94. IEEE Computer Society. ISBN 0-8186-6310-3, 1994 pp. 71–80. doi:10.1109/LICS.1994.316085.

[31] Nielsen M, Palamidessi C, Valencia FD. On the expressive power of temporal concurrent constraint program. languages. In: Proc. of PPDP'02. ACM, 2002 pp. 156–167. doi:10.1145/571157.571173.

[32] Olarte C, Pimentel E. On concurrent behaviors and focusing in linear logic. *Theor. Comput. Sci.*, 2017. **685**:46–64. doi:10.1016/j.tcs.2016.08.026.

[33] Olarte C, Pimentel E, Rueda C. A concurrent constraint programming interpretation of access permissions. *TPLP*, 2018. **18**(2):252–295. doi:10.1017/S1471068418000017.

[34] Chemillier M. Les Mathématiques Naturelles. Odile Jacob, 2007.

[35] Olarte C, Rueda C, Sarria G, Toro M, Valencia FD. Concurrent Constraints Models of Music Interaction. In: Assayag G, Truchet C (eds.), Constraint Programming in Music, pp. 133–153. Wiley, 2011.

[36] Guzmán M, Haar S, Perchy S, Rueda C, Valencia FD. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *J. Log. Algebr. Meth. Program.*, 2017. **86**(1):107–133. doi:10.1016/j.jlamp.2016.09.001.

[37] Falaschi M, Olarte C. An assertion language for slicing Constraint Logic Languages. In: Mesnard F, Stuckey P (eds.), Proceedings of LOPSTR'2018, volume 11408 of *LNCS*. Springer, Berlin, Heidelberg, 2019 pp. 148–165.

[38] Jaffar J, Maher MJ, Marriott K, Stuckey PJ. The Semantics of Constraint Logic Programs. *J. Log. Program.*, 1998. **37**(1-3):1–46. doi:10.1016/S0743-1066(98)10002-X.

[39] Falaschi M, Olarte C, Palamidessi C, Valencia F. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In: Dahl V, Niemelä I (eds.), Proc. of ICLP 2007, volume 4670 of *LNCS*. Springer, 2007 pp. 271–285.

[40] Bodei C, Brodo L, Gori R, Levi F, Bernini A, Hermith D. A static analysis for Brane Calculi providing global occurrence counting information. *Theoretical Computer Science*, 2017. **696**:11–51.

[41] Bodei C, Brodo L, Gori R, Hermith D, Levi F. A Global Occurrence Counting Analysis for Brane Calculi. In: Proc. of LOPSTR 2015, volume 9527 of *Lecture Notes in Computer Science*. Springer, 2015 pp. 179–200.

[42] Bodei C, Brodo L, Focardi R. Static Evidences for Attack Reconstruction. In: Proc. of Programming Languages with Applications to Biology and Security, volume 9465 of *Lecture Notes in Computer Science*. Springer, 2015 pp. 162–182.

[43] Olarte C, Chiarugi D, Falaschi M, Hermith D. A proof theoretic view of spatial and temporal dependencies in biochemical systems. *Theor. Comput. Sci.*, 2016. **641**:25–42.

[44] Chiarugi D, Falaschi M, Hermith D, Olarte C, Torella L. Modelling non-Markovian dynamics in biochemical reactions. *BMC Systems Biology*, 2015. **9**(S-3):S8.

[45] Bernini A, Brodo L, Degano P, Falaschi M, Hermith D. Process calculi for biological processes. *Natural Computing*, 2018. **17**(2):345–373.

[46] Bodei C, Brodo L, Bruni R, Chiarugi D. A flat process calculus for nested membrane interactions. *Scientific Annals of Computer Science*, 2014. **24**(1):91–136.